Uma Proposta de Múltiplas Visões Coordenadas para Apoiar Análise de Impacto de Mudança

Fernanda Madeiral Delfim[1], **Rogério Eduardo Garcia**[1]

[1]Departamento de Matemática e Computação
Faculdade de Ciências e Tecnologia
Universidade Estadual Paulista "Júlio de Mesquita Filho" (UNESP)
Presidente Prudente – SP, Brasil
fer.madeiral@gmail.com, rogerio@fct.unesp.br

Abstract. *Program Comprehension is an essential task in Software Maintenance. To make a modification, it is necessary to understand the existing program to be maintained not only to make the modification itself, but also to analyze and to identify code fragments that can be impacted by modification. In this paper we present a visual approach that includes six coordinated views to support program comprehension and change impact analysis by visual exploration of different levels of detail. In addition, is shown the tool in which the proposed visual approach was implemented, with their results (visual presentations), and a discussion of its use.*

Resumo. *Compreensão de Programa é uma tarefa fundamental na Manutenção de Software. Ao realizar uma manutenção, é necessário entender o programa existente não somente para efetuar as alterações em si, mas também para analisar e identificar trechos de código que podem ser impactados pela alteração. Neste trabalho é apresentada uma abordagem visual que inclui seis visões coordenadas para apoiar compreensão de programa e análise de impacto de mudança pela exploração visual de diferentes níveis de detalhe. Além disso, é apresentada a ferramenta na qual foi implementada a abordagem visual proposta, juntamente com os seus resultados (representações visuais), e uma discussão sobre seu uso.*

1. Introdução

A manutenção de software implica na compreensão de software antes da realização de qualquer modificação. Compreender um sistema de software consiste em uma atividade cognitiva complexa. É preciso entender o software em seus diversos níveis de abstração, sua arquitetura e sua estrutura [Novais et al. 2011], não somente para realizar uma mudança em si, mas também para analisar o impacto da mudança e para tomar decisões.

A *análise de impacto de mudança* permite que as unidades potencialmente afetadas (linhas de código e/ou unidades modulares) sejam conhecidas antes das alterações, evitando o aumento no trabalho intensivo e propenso a erros, bem como reduzindo os riscos de iniciar uma mudança cara [Mohamad et al. 2007]. Mudanças podem ser simples e localizadas, mas também podem consistir em alterações significativas em um código fonte. Para estas, é preciso compreender o programa de modo a permitir a análise do impacto da modificação antes de realizá-la. De acordo com Mohamad et al. [2007], a capacidade de visualizar as dependências entre unidades de código pode ajudar um mantenedor

determinar as ações adequadas no que diz respeito à tomada de decisões. Portanto, o uso de Visualização de Software pode ser útil para observar o código que pode ser impactado por mudanças, ajudando, assim, a compreensão do programa [Mohamad et al. 2007].

Neste trabalho é proposta uma abordagem visual para apoiar a compreensão de programas orientados a objetos (implementados em Java) e a investigação do impacto de mudança baseada em associações/dependências entre unidades modulares e entre instruções. A abordagem proposta inclui seis apresentações visuais coordenadas para permitir analisar: 1) a estrutura de programa; 2.1) as associações baseadas em chamadas de métodos em nível de classe 2.2) e em nível de método; 3) as dependências de controle e de dados de instruções de programa; 4) como um conjunto de instruções (fatias) é distribuído em classes; 5) o bytecode de programa. A ferramenta de visualização de software, nomeada $SoftVis_{4CA}$ (*Software Visualization for Code Analysis*), foi desenvolvida para apoiar a abordagem visual proposta.

Para apresentar a proposta, este artigo encontra-se organizado como segue: na Seção 2 são apresentados a motivação e os trabalhos relacionados; na Seção 3 é apresentada a abordagem visual proposta; na Seção 4 é apresentada uma discussão sobre a abordagem; na Seção 5 são apresentadas as considerações finais.

2. Motivação e Trabalhos Relacionados

Considerando o código-fonte, é possível restringir as mudanças para as operações de inserção, exclusão e alteração de linhas de código. No entanto, qualquer mudança pode influenciar (impactar) outras linhas de código, podendo ser localizada (restrita a uma unidade de código) ou podendo influenciar em mais de uma unidade de código e, assim, requer mudanças em outras unidades (propagação de mudança) [Buckley et al. 2005]. Independentemente da influência de uma mudança, é necessária a compreensão de como o software está implementado, para determinar quais elementos do sistema são relevantes para a modificação e para entender quais elementos podem ser afetados pela modificação (análise de impacto e de propagação de mudança).

Técnicas de análise de impacto de mudança têm sido propostas na literatura, usando diversos meios para identificar as partes do programa que podem ser afetados, como grafos de chamadas, análise de dependência, medição de acoplamento entre unidades e fatiamento de programa [Li et al. 2012]. Ferramentas têm sido propostas para apoiar tais técnicas, três delas apresentadas brevemente a seguir.

JTracker [Gwizdala et al. 2003] é uma ferramenta para auxiliar na propagação de mudança, de modo que quando o programador modificar o código de uma classe, as classes vizinhas potencialmente impactadas são marcadas para inspeção pelo programador. *EAT* [Apiwattanapong et al. 2005] utiliza informação dinâmica e analisa sequências de execução para gerar resultados em nível de método, de acordo com a ideia de que todas as entidades que são executadas depois de uma entidade modificada são potencialmente afetadas pela modificação. *Columbus* [Ferenc et al. 2002] é uma ferramenta que calcula medidas de acoplamento estrutural entre classes de um programa. De acordo com os resultados dessas medidas, os elementos que tem um alto acoplamento com os elementos a serem alterados são indicados como mais prováveis a serem impactados pela mudança.

As ferramentas existentes nas quais são implementadas técnicas de análise de

impacto de mudança geram resultados que devem ser compreendidos por um usuário[1], como medidas de acoplamento e unidades de programa potencialmente afetadas (classes, métodos e trechos de código). Sendo assim, o papel do usuário é importante para a compreensão do programa e para a análise de impacto de mudança.

Adicionalmente, [LaToza et al. 2007] concluíram que a compreensão de programa é conduzido por *fatos* ao invés de instruções de código. Alguns exemplos desses fatos são: se um método específico tem um efeito no seu contexto; se um método específico é chamado muitas vezes; ou se alguns métodos são chamados após um evento específico. Ferramentas que externalizam fatos podem ajudar a manutenção, focando os desenvolvedores diretamente nos fatos relevantes e reduzindo o tempo que eles gastam para a compreensão de programa. Técnicas e ferramentas de Visualização de Software podem ser úteis para a compreensão de programa e para a análise de impacto de mudança, permitindo ao usuário uma exploração visual de características e de fatos de um programa.

3. O Modelo de Coordenação Proposto

Neste trabalho é proposta uma abordagem visual para apoiar a compreensão de programa e a análise de impacto de mudança, implementada em uma ferramenta de Visualização de Software *desktop*, nomeada $SoftVis_{4CA}$. A arquitetura da $SoftVis_{4CA}$ é organizada em três camadas como retratada na Figura 1.

A abordagem proposta consiste em um conjunto de apresentações visuais – *Visão Hiperbólica em Nível de Classe*, *Visão Hiperbólica em Nível de Método*, *Visão de Agregados*, *Visão Treemap*, *Visão de Barras e Listras* e *Visão de Bytecode* – organizadas na *Camada de Visualização* (Figura 1) para mostrar diferentes características de programas. Usando um bytecode de programa como entrada, na *Camada de Dados* são realizadas análises estática e dinâmica para coletar informações do código, mantidas em estruturas de dados usadas para gerar as apresentações visuais na *Camada de Visualização* e para coordená-las pela *Camada de Controle*.

As cores dos itens visuais são pré-definidas em cada apresentação visual, mas o usuário pode modificar tais cores. Adicionalmente, existe um mapeamento de cores baseado em chamadas de métodos usado nas visões – para cada unidade (i.e., pacotes, classes e métodos) é computado o valor da métrica *fan-in*, e um gradiente de cor a partir de cinza claro para vermelho é usado para representar os valores obtidos (vermelho representa o valor maior). É possível filtrar unidades de código com base em valor de *fan-in* por meio da definição de um limiar e observar ele nas projeções e, assim, é possível visualizar as unidades com alta frequência de chamadas (unidades fortemente acopladas às outras).

Os resultados obtidos com a $SoftVis_{4CA}$ são mostrados usando um programa orientado a objetos de simulação de elevador [Do et al. 2005]. Nas Figuras 2(a) e 2(b) são mostradas duas projeções hiperbólicas baseadas em chamadas de métodos: a primeira em nível de classe e a segunda em nível de método. Os nós representam unidades (classes ou métodos), e as arestas direcionadas representam as chamadas entre elas. A projeção hiperbólica mostra um nó em foco (escolhido pelo usuário), maior do que os outros e posicionado no centro do espaço de projeção. Os nós vizinhos do nó em foco são coloridos

[1] *Usuário* refere-se ao profissional que utiliza as ferramentas de análise de impacto de mudança.

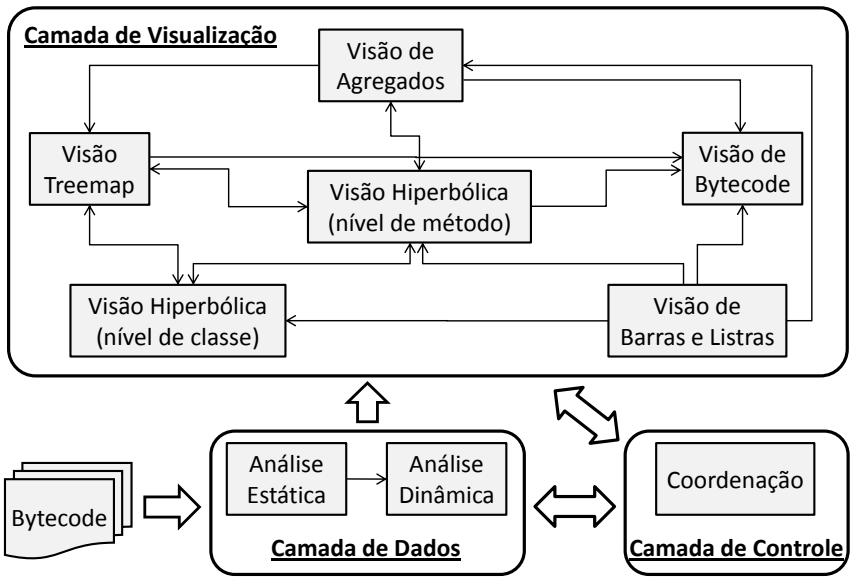
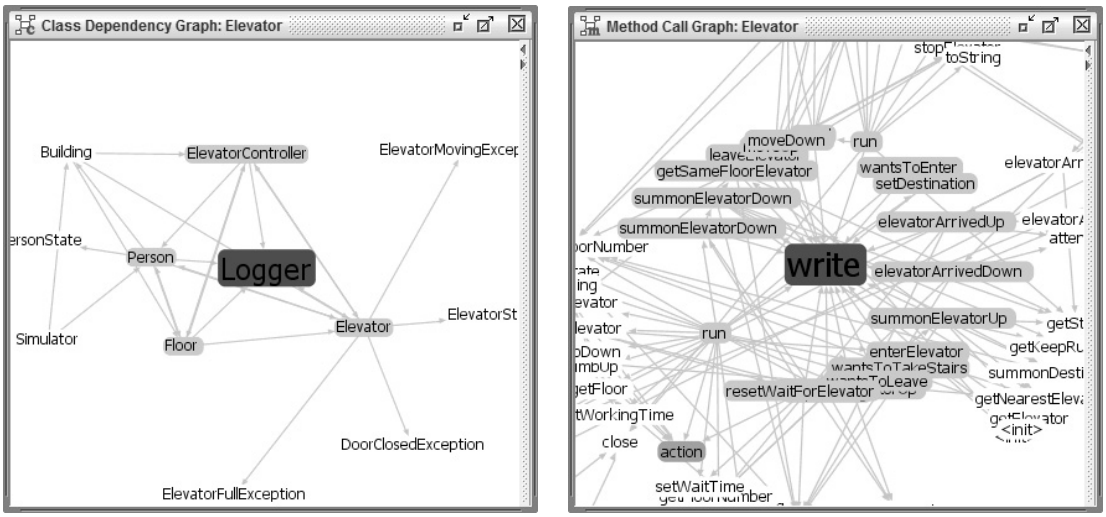


Figura 1. Arquitetura da $SoftVis_{4CA}$.

em rosa, e os outros nós não são coloridos. Tais visões ajudam a tornar explícito as unidades que são mais frequentemente chamadas – a classe `Logger` e os métodos `write` e `action` estão coloridos usando o gradiente de cor baseado na métrica *fan-in*.



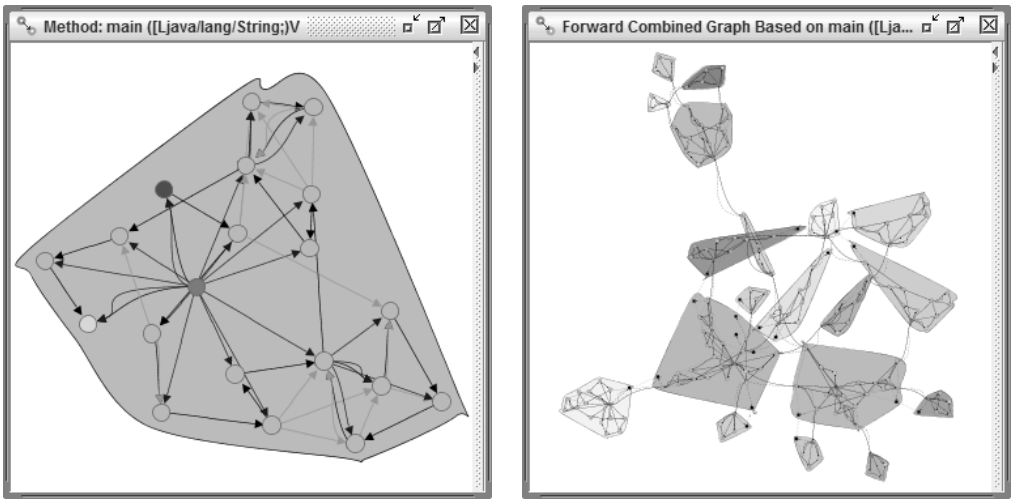
(a) Em Nível de Classe. (b) Em Nível de Método.

Figura 2. *Visões Hiperbólicas*.

A *Visão de Agregados* mostra grafos em nível de instruções de programa. Cada método do programa tem seu próprio grafo, e grafos podem ser combinados por chamadas de métodos a partir de ou para um método selecionado (*backward* ou *forward*). Nas Figuras 3(a) e 3(b) são mostrados os dois tipos de visão: visão do grafo de um método (intra-método) e visão de um grafo combinado (inter-métodos), respectivamente. Existem dois tipos de nós nessas visões: nós físicos e os nós virtuais. Os nós físicos representam instruções bytecode de programa (coloridos em rosa). Os nós virtuais são introduzidos

para representar entradas de métodos (coloridos em verde escuro), saídas de métodos (coloridos em verde claro) ou parâmetros (coloridos em preto). As arestas representam fluxo de controle (linhas contínuas coloridas em preto), exceção (linhas tracejadas coloridas em preto), dependência de controle (linhas contínuas representam intra-método e as linhas tracejadas representam inter-método, ambas coloridas em azul) ou dependência de dados (linhas contínuas representam intra-método e as linhas tracejadas representam inter-método, ambas coloridas em verde). Além disso, os nós são agregados por métodos, representados por polígonos curvo (nomeado *região de agregado*) coloridos por uma cor específica para cada método (foi usado transparência por causa de sobreposição).

Fatias podem ser visualizadas e, por interação com a *Visão de Agregados*, o usuário pode escolher um item visual que representa uma instrução em específico (critério de fatia) e o tipo de fatia (estática/dinâmica e *backward*/*forward*). Assim, uma outra *Visão de Agregados* é criada para apresentar a fatia usando os parâmetros escolhidos.



(a) Intra-Método. (b) Inter-método.

Figura 3. *Visões de Agregados*.

A *Visão Treemap* apresenta hierarquicamente a estrutura de programa por retângulos aninhados. O retângulo externo representa o programa todo (raiz), e os retângulos internos representam pacotes, classes, métodos e casos de testes. Os retângulos são coloridos usando um gradiente linear simples de cinza (cinza escuro representa o programa todo). Cada retângulo folha representa um método e o seu tamanho é proporcional a quantidade de instruções bytecode do método. Na Figura 4 está destacado o método selecionado na Figura 2(b) (sua classe em vermelho) e os métodos relacionados em rosa.

Na Figura 5 é mostrada a *Visão de Barras e Listras*. As barras representam classes – a altura de cada barra é proporcional a quantidade de instruções bytecode da classe que ela representa. As listras representam instruções de programa usando diferentes cores atribuídas para cada fatia. Tal visão provê uma visão geral de como uma fatia é distribuída em múltiplas classes.

Além disso, a $SoftVis_{4CA}$ permite visualizar instruções bytecode usando duas rolagens sincronizadas: rolagem da direita para visão geral e rolagem da esquerda para detalhes. Na rolagem da direita os retângulos representam um conjunto de instruções

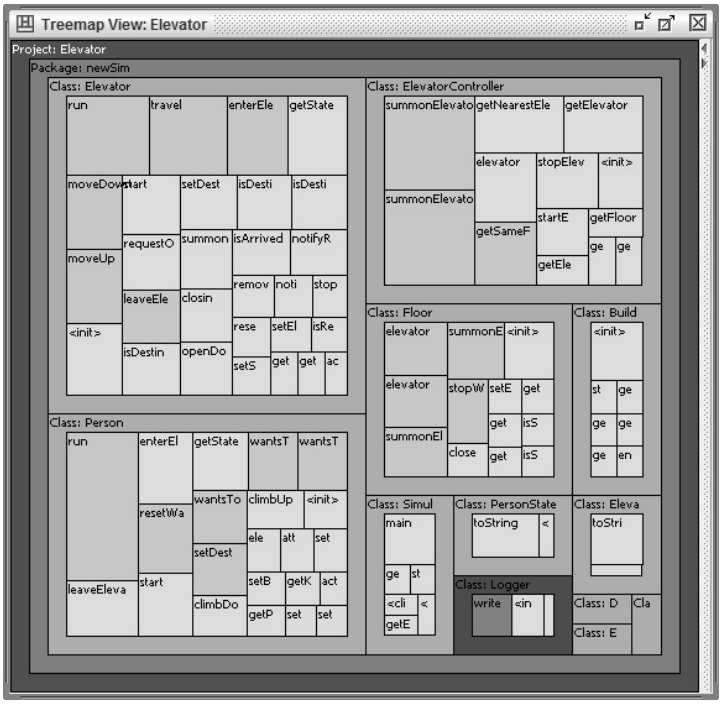


Figura 4. *Visão Treemap*.

agrupadas por *labels* do bytecode do programa. Na rolagem da esquerda são mostradas as instruções bytecode de retângulos selecionados na rolagem da direita, agrupadas pelo método que as possuem. Similar à *Visão de Agregados*, cada método é mostrado usando uma cor diferente – na Figura 6 é mostrado o método `Logger.write (Ljava/lang/String;)V` representado em vermelho e outro método é representado em azul. O mapeamento de cores é o mesmo da *Visão de Agregados*.

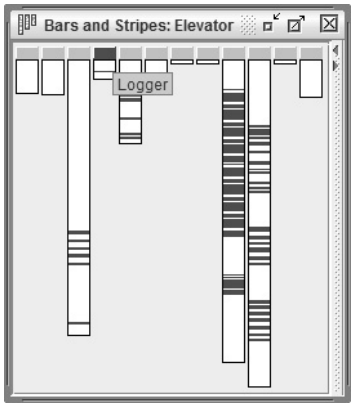


Figura 5. *Visão de Barras e Listras*.

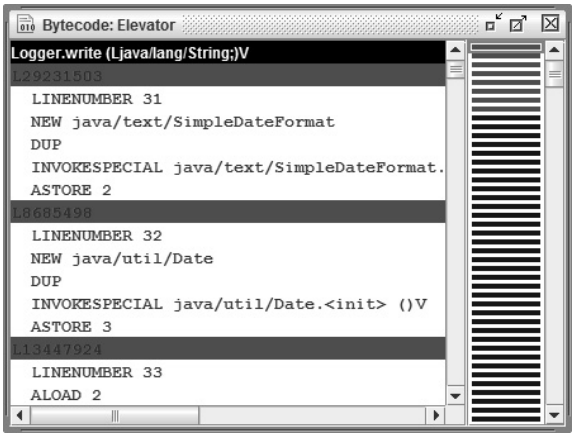


Figura 6. *Visão de Bytecode*.

4. Discussão

A compreensão de programa para análise de impacto e de propagação de mudança deve ser realizada em diferentes níveis de abstração, considerando as possíveis mudanças que podem ser realizadas em sistemas. A abordagem visual proposta apoia tal compreensão

pela coordenação de seis apresentações visuais de diferentes aspectos de um programa. A seguir são apresentados os seguintes cenários que ressaltam o uso das visualizações propostas: modificação da interface de unidades, modificação interna em unidades, e modificação da estrutura de programa.

Na manutenção, a alteração de métodos pode afetar diretamente ou indiretamente outros métodos – por exemplo, qualquer modificação na interface de um método implica na necessidade de alteração das instruções de chamadas para o método modificado. A análise de chamadas de um sistema é importante, podendo ajudar a avaliar o impacto (e a extensão) de uma alteração [Lehnert 2011]. As *Visões Hiperbólicas* permitem tal análise de chamadas para avaliar impacto e propagação de mudança em nível de classe e de método. Na *Visão Hiperbólica em Nível de Método*, por exemplo, é possível observar métodos a serem inspecionados para adequar à chamada de um método modificado. Ressalta-se que é possível observar nas *Visões Hiperbólicas* o acoplamento entre classes e entre métodos por meio do gradiente de cores baseado na métrica *fan-in*.

A mudança intra-método, tanto de um dado (e seu tipo) como de uma computação, pode impactar em diferentes instruções executadas após o ponto de modificação, pela influência de fluxo de controle e de dados. Se a computação de uma variável for alterada, um ponto importante é observar as possíveis instruções que podem ser afetadas (i.e., as instruções que usam o valor da variável alterada após sua computação), diretamente e indiretamente. Na ferramenta $SoftVis_{4CA}$ isso é possível analisando os grafos de instrução e as fatias na *Visão de Agregados*, permitindo a análise de dependência de controle e de dados, para frente e para trás (*forward* ou *backward*). Adicionalmente, de acordo com [Ishio et al. 2012], as relações entre os fragmentos de código devem ser visualizadas de maneira que os desenvolvedores possam selecionar uma parte de código fonte apropriada para ser investigada. Tal recurso é oferecido pela ferramenta, permitindo que a análise seja localizada em regiões de interesse (representação de partes de código).

Modificações na estrutura de programa podem ser necessárias. Se for necessária a criação de uma nova classe (seja ela nova ou gerada pela refatoração de uma classe existente), por exemplo, é preciso acomodá-la apropriadamente em algum pacote. Para a adição e exclusão de unidades (pacotes, classes e métodos), bem como para movê-las, é necessária a visão geral da estrutura do programa. Para isso, a ferramenta provê a visão geral da estrutura de um programa utilizando *Treemap*, que auxilia a tomada de decisões quanto a modificações relacionadas à estrutura do programa.

5. Considerações Finais

Neste trabalho é apresentada uma abordagem visual e sua implementação ($SoftVis_{4CA}$), que consiste em coordenar múltiplas visões usando um mapeamento de cores baseado na métrica *fan-in* e uma visualização de fatias, permitindo: 1) exploração visual de software em diferentes níveis de detalhe; 2) visualização dos métodos mais frequentemente chamados e com forte acoplamento aos outros; 3) análise de dependência de controle e de dados.

Os principais objetivos são apoiar a compreensão de programas orientados a objetos implementados em Java e a análise de impacto e de propagação de mudanças, para apoiar a manutenção de software. A ferramenta têm sido avaliada, e pretendemos realizar um experimento controlado para avaliar a abordagem proposta.

Referências

Apiwattanapong, T., Orso, A., and Harrold, M. J. (2005). Efficient and Precise Dynamic Impact Analysis Using Execute-After Sequences. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, p. 432–441, New York, NY, USA. ACM.

Buckley, J., Mens, T., Zenger, M., Rashid, A., and Kniesel, G. (2005). Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309–332.

Do, H., Elbaum, S. G., and Rothermel, G. (2005). Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435.

Ferenc, R., Beszédes, Á., Tarkiainen, M., and Gyimóthy, T. (2002). Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM '02)*, p. 172–181. IEEE Computer Society.

Gwizdala, S., Jiang, Y., and Rajlich, V. (2003). JTracker - A Tool for Change Propagation in Java. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR '03)*, p. 223–229, Washington, DC, USA. IEEE Computer Society.

Ishio, T., Etsuda, S., and Inoue, K. (2012). A Lightweight Visualization of Interprocedural Data-Flow Paths for Source Code Reading. In *Proceedings of the 20th International Conference on Program Comprehension (ICPC '12)*, p. 37–46.

LaToza, T. D., Garlan, D., Herbsleb, J. D., and Myers, B. A. (2007). Program Comprehension as Fact Finding. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '07)*, p. 361–370, New York, NY, USA. ACM.

Lehnert, S. (2011). A Review of Software Change Impact Analysis. Technical report, Ilmenau University of Technology, Department of Software Systems / Process Informatics.

Li, B., Sun, X., Leung, H., and Zhang, S. (2012). A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*.

Mohamad, R. N. B., Idris, N. B. B., and Ibrahim, S. B. (2007). A Change Impact Analysis to Support Program Understanding Using Visualization Method. In *Postgraduate Annual Research Seminar (PARS '07)*.

Novais, R. L., de Figueiredo Carneiro, G., Júnior, P. R. M. S., and de Mendonça Neto, M. G. (2011). On the Use of Software Visualization to Analyze Software Evolution – An Interactive Differential Approach. In Zhang, R., Cordeiro, J., Li, X., Zhang, Z., and Zhang, J., editors, *Proceedings of the 13th International Conference on Enterprise Information Systems (ICEIS '11)*, p. 15–24. SciTePress.